ARM Command Line Workbook

CS160 Computer Organization
Version 1.1
October 27th, 2002
Revised Fall 2005

ARM University Program
Version 1.0
January 14, 1997

Introduction

Aim

This workbook provides the student with a basic understanding of the facilities provided by the ARM Toolkit for command line users.

Note: For all of the command line tools provided with the toolkit it is possible to get on-line help on the command line parameters by specifying the toolname followed by *-help*.

Prerequisites

Before getting started, you will need to add the ARM tools to your command path. Make the following change to the `~/.zsh_files/.zshrc` file:

- `export PATH="$PATH:/pkgs/arm202u/bin"`

Next, you will need to add an environment variable so that the ARM compiler knows where to find the appropriate libraries. Make the following change to the `~/.zshrc` file:

- `export ARMLIB="/pkgs/arm202u/lib"`

Ask the GTAs for help if you don't know how to do this. When done, issue the commands:

```
source  ~/.zshrc
~cs160/bin/armsetup.pl
```

The latter will create a hierarchy of subdirectories and files in a directory called `arm` within your `cs160` directory. The reference `/cmdline/hello.c` thus should be read as `~/arm/cmdline/hello.c`

Compiler

The 'Hello World' example program is a simple C program which calls a subroutine. This program file can be found as `/cmdline/hello.c`

```
#include <stdio.h>

/* Declare subroutine before used by main */
void subroutine (void);

int main() {
  printf("Hello World from main\n");
  subroutine();
  printf("And Goodbye from main\n");
  return 0;
}

/* Define subroutine */
void subroutine() {
  printf("Hello from subroutine\n");
}
```

Exercise 1.1

Compile this program with the ARM C compiler type

```
armcc -g hello.c
```

The C compiler automatically invokes the linker to produce an executable with the same name as the C module but with no file extension.

Thus this command will compile the C code, link with the default C library and produce an ARM Image Format (AIF) executable called `hello`.

Exercise 1.2

To specify a differently named executable then use the *-o* compiler option

```
armcc -g hello.c -o tester
```

Note: The *-g* option adds high-level debugging information to the final object. If -g is not specified then the program will still produce the same results when executed but it will not be possible to see the source code in the debugger. This is discussed in more detail later.

To execute this program type

```
armsd hello
go
```

`armsd` responds with

```
Hello World from main
Hello from subroutine
And Goodbye from main
Program terminated normally at PC = 0x00008da8
(__rt_exit + 0x24)
+0024 0x00008da8: 0xef000011  .... :    swi      0x11
```

Exercise 1.3

It is possible to produce an assembly language listing generated by the C compiler using the *-S* option. (Note : The *-S* is an upper case character.)

```
armcc -S hello.c
```

This command will produce an output file called `hello.s` which can be viewed using any text editor.

Multiple C Files

If multiple C files are used in an application, it is necessary to force the C compiler to simply produce a `.o` file (and not invoke the linker).

Exercise 1.6

Produce an object file only using the C compiler by typing

```
armcc -g -c hello.c
```

Look at the directory listing and ensure that a file named `hello.o` has recently been written.

Exercise 1.7

Create two new modules from the original `hello.c` program. The first module should contain all the original code but not the subroutine procedure. (Hint: You will need to declare the subroutine as an external procedure in the first module.) The second module should only contain the subroutine procedure.

(Hint: Two example modules called `hello1.c` and `hello2.c` are provided with the workbook with the required modifications already completed.)

Now compile these two new modules using the ARM compiler.

```
armcc -g -c hello1.c hello2.c
```

Exercise 1.8

It is now necessary to link your two code modules with the ARM C library which can be found in the `lib` subdirectory of the toolkit installation. Thus, type:

```
armlink hello1.o hello2.o $ARMLIB/armlib.32l -o demo
```

where the l (lowercase L) suffix indicates a little-endian version of the library. There is also a big-endian version called `armlib.32b`.

The example links two source files with the 32-bit version of the C library and produces an executable output file called `demo`.

Exercise 1.9

Now load and run the demo program and see that the output is the same as that shown in Exercise 1.2.

Assembler

Practical experience of ARM assembler command line usage is detailed in the ARM Assembler Workbook. The ARM assembler is called `armasm.`

Debugger

The ARM symbolic debugger can be invoked from the command line by simply typing `armsd`. The user will then be greeted by the specifics of their version of the debugger followed by the `armsd` prompt.

It is possible to get on-line help from the debugger by issuing the `help` command. Further detail is available by specifying `help` followed by the particular help subject, for example, `help break` would provide a detailed description of the syntax and usage of the `break` command.

Hint: It is possible to shorten most of the commands used in the debugger. Issuing the `help` command displays the full list of available commands, with the minimum number of characters required for the command displayed in upper case.

Loading an executable into the debugger.

It is possible to specify the name of the executable being debugged upon invocation of the debugger.

```
armsd hello
```

Alternatively the executable can be loaded after the debugger had been started by using the `load` command.

```
armsd
load hello
```

To quit from the debugger issue the `quit` command.

```
quit
```

Viewing Source Code

To view source code it is necessary to have compiled or assembled the source code with the -g option.

```
armcc -g hello.c -o hello
armsd hello
```

Set a breakpoint on the first statement in function main and execute the program up to that point.

```
break main ; go
```

Now view the C source.

```
type
```

It is possible to list the C source from a particular line number.

```
type 1
```

With the `type` command an error is reported if program execution is not currently within a source code module. (The `reload` command will reload the object file specified on the `armsd` command line, or the last `load` command. Breakpoints (but not watchpoints) remain set after a `reload` command.)

```
reload
type
```

`armsd` responds with

```
** Error: No file
```

It is possible to view the source code separately from within execution context by specifying the module filename and a start and end line.

```
type 1,10,hello.c
```

However, it is not necessary to specify the start and end lines to display.

```
type ,,hello.c
type 1,,hello.c
```

If the source code is in another directory then it is necessary to include either the full pathname or the relative pathname from the current directory (in this case `~/arm`)

```
type 1,30,/cmdline/hello.c
type 1,25,cmdline/hello.c
```

Viewing assembler source code uses the same debugger command syntax.

Breakpoints

To view the current set of breakpoints the `break` command is issued without any parameters.

```
break
```

`armsd` responds with

```
#1 hw break main:7
```

To set a breakpoint on a procedure then the procedure name is required.

```
break main
```

It is also possible to set a breakpoint on a procedure exit.

```
break main:$exit
```

To set a breakpoint on a particular line within a source file it is possible to either specify the procedure name or the module name (preceded by a #) followed by the module line number. (The following two commands will set a breakpoint on the same statement.)

```
break main:7
break #hello:7
```

If no compiled code exists for the source line then the debugger flags up the error.

```
** Error: No code compiled for source line
```

Further Debugger Facilities

The following sections of this workbook also make use of another example program called `trial.c`. The program serves no real function, but simply acts as an example which can be used to set watchpoints and breakpoints on.

The `initmem` function is used to clear a block of memory.
The `quickfunc` function is a very simple function.
The `results` function is a demonstration of semi-hosting.

This file can be found at `/cmdline/trial.c`

```
#include <stdio.h>

#define BLOCKSTART (int *) 0x4000
#define BLOCKEND   (int *) 0x4100
#define TESTADDR   (int *) 0x4200


/* Memory initialization to zero */
static void initmem() {
  int *i;
  int a;

  a = 0;
  i=BLOCKSTART;

  do {
    *i++ = 0;
    a++;
  } while (i<BLOCKEND);
}

static void results() {
  printf("The memory area has now been cleared.\n");
}

static void quickfunc (int a, int b, int c) {
  int x,y,z;
  x = 3;
  y = 4;
  z = x * y;
  x = (z << 8) + y;
}

int main() {
  int *datablock;
  int a,b,c;

  datablock = BLOCKSTART;
  a = 10;
  b = 20;
  c = 30;

  initmem();
  quickfunc(a,b,c);
  quickfunc(b,c,a);
  quickfunc(c,a,b);
  results();
}
```

Complex breakpoints.

It is possible to specify the number of times a statement is to be executed before the application is suspended, the default of which is one (1).

```
armsd trial
break initmem:28 10
go
```

`armsd` responds with

```
Breakpoint #1 at #trial:initmem, line 28 of trial.c
   28   *I++ = 0;
```

At this point the code statement displayed after the breakpoint has not been executed and will be the next statement to be executed.

To execute a debugger command (or commands) after a breakpoint has been reached then the commands must be enclosed in braces and follow the breakpoint definition.
(The `where` command displays the program position when the breakpoint is executed.)

```
break initmem:28 10 do {where; print a}
go
```

The breakpoint can also be conditionally defined by using the `if` clause. (The following will stop program execution at line 28 of the module containing the `initmem` procedure, display the current program location and value of `a`, only if local variable `a` is greater than ten.)

```
break initmem:28 10 do {where; print a} if a > 10
go
```

To remove a breakpoint the number of the breakpoint must be specified preceded by a #

```
unbreak #1
```

Note: Breakpoints are not renumbered following deletion of other breakpoints. Once a breakpoint has been assigned a number, the original number is retained.

Viewing Memory Contents

The `list` command displays memory locations in instruction, hexadecimal, and character format. It is possible to specify the size of the instruction followed by a start location and byte count. If the size is not defined then the debugger will attempt to guess the instruction size from the debugging information.

```
armsd hello
list 0x0
list @main,20
list 0x8000
```

Note: Low-level symbols are differentiated from high-level ones by preceding them by an at-sign “@”. A low-level symbol for a procedure refers to its call address, which is often the fist instruction of the stack frame initialization. This differs from a high-level symbol for a procedure which refers to the address of the code generated by the first statement in the procedure.

If source level debugging is enabled (by using the -g option) then it is not necessary to use the @ symbol.

```
list subroutine,100
```

Examining Memory Locations

To view the contents of memory the `examine` command displays both hexadecimal and ASCII formats between a pair of addresses. (The following will, by default, display 128 bytes of memory.)

```
armsd trial
examine 0x4000
```

To view a larger section of memory then either it is possible to specify a number of bytes to be displayed, or an end address.

```
examine 0x4000,+300
examine 0x4000,+0x300
examine 0x4000,0x4500
```

Watchpoints

To set watchpoints on memory locations and variables use the `watch` command. (The following commands set two watchpoints: The first is on the local variable `a` in the `initmem` procedure, and the second on the memory location `0x4000`.)

```
armsd trial
break initmem
go
watch 0x4000
watch a
go
```

`armsd` responds with

```
Watchpoint #2 at a changed at #trial:initmem, line 24 of trial.c
          24 a  =  0;
```

Unlike breakpoints the statement displayed when a watchpoint is taken has been executed, and was the statement that caused the change to the variable or memory location.

To view the watchpoints which have been set simply type `watch` without parameters.

```
watch
```

`armsd` responds with

```
#1 sw at 0x4000
#2 sw at a
```

Using these watchpoint numbers it is possible to delete the watchpoint.

```
unwatch #2
```

Complex Watchpoints

Watchpoints can be conditionally executed and perform debugger commands in the same way as breakpoints. (The following will stop program execution when the value in local variable `a` has changed twenty times, only if the value in `a` is less than 100.)

```
watch a 20 do {print a} if a < 100
```

Accessing variables, registers and arguments

The `print` command examines the contents of the application's variables, or displays the result of arbitrary calculations involving variables and constants. It is possible to change the default display format by specifying a display format immediately after the `print` command.

```
print a
print/%x a
print BLOCKEND
print/%d BLOCKEND
print/%x a * BLOCKEND
print I
```

To view a list of local variables (symbols) from within a procedure, type

```
symbols
```

To view a list of the global variables, the module name must be specified.

```
symbols trial
```

To view a list of procedure parameters from outside the execution context, the procedure name must be specified.

```
symbols quickfunc
```

It is possible to view all of the registers.

```
reg
```

This will display the register `r0` to r15 and the CPSR. The CPSR shows the current status flags, the interrupt-disable flags, and a textual description of the current mode. The flag characters in lowercase signify that the flag bit is *clear*; likewise, characters in uppercase signify that the bit is *set*.

To view the banked registers for a different mode then the mode must be specified.

```
reg IRQ32
```

Typical mode definitions are `USER32`, `SVC32`, `IRQ32`, `FIQ32`, `ABORT32`, `SYSTEM32`, and `UNDEF32`. The textual description for the mode can be in uppercase or lowercase.

During debugging it is often necessary to change the value stored in a variable or register. This is possible using the `let` command (although it is not necessary to actually specify `let`.)

```
reload ; go
let r0 = 10
reg
let a = 20
print a
a = 25
print a
```

Similarly, it is possible to change to value of the CPSR. (The following will set the Negative and Zero conditional flags in the CPSR and set the mode to `FIQ` mode.)

```
cpsr = %NZ_fiq32
```

Similarly it is possible to modify the value of the SPSR when it a privileged mode.

```
spsr = %IF_IRQ32
```

To view the values of any arguments passed into a procedure from anywhere within the execution context of the procedure use the `arguments` command.

```
quit
armsd trial
break quickfunc
go
arguments
```

Stepping through applications

The `step` command steps through one of more application statements. To actually step into a procedure from the calling line, `step in` must be specified. (The following instruction sequence steps through the local variable initializations and then steps into the `initmem` procedure.)

```
quit
armsd trial
break main
go
step 5
step in
```

To step back out of a procedure (to the line of code which follows the originating procedure call) type

```
step out
```

To step through instructions (as opposed to statements), set the language to `none` or use the `istep` command

```
reload ; go
language none
step
```

When using the `istep` command the debugger will still only display the source code and will display the same source line if several instructions are associated with it. To view each step the `list` command must be issued. It is still necessary to use `istep in` to follow a branch-with-link instruction. (Note that after a `reload` the language setting is reset.)

```
reload; go
istep 4
istep in
istep
list
istep ; list
```

To see what the language setting is type

```
language
```

To actually display source code statements interleaved with ARM assembler instructions, type

```
istep in; list ^pc,+4
```

This is quite a verbose command to have to type and retype, so it is best to use the `alias` command and assign the full command to, for example, an unused character.

```
alias f "istep in; list ^pc,+4"
```

From then on simply typing `f` will perform the desired command.

To list the aliases that have been set up, type `alias` without parameters.

DECAOF - Decode AOF

When a ARM Object Format object has been produced by the compiler or assembler it is possible to decode this and examine the output. (The following will produce decoded output, showing disassembled code areas, from the file `hello.o`)

```
decaof -c hello.o
```

The output can be directed to another file so that it can be viewed by any text editor.

```
decaof -c hello.o > hello.decaof
```